POCKETSPHINX: A FREE, REAL-TIME CONTINUOUS SPEECH RECOGNITION SYSTEM FOR HAND-HELD DEVICES

David Huggins-Daines, Mohit Kumar, Arthur Chan,
Alan W Black, Mosur Ravishankar, and Alex I. Rudnicky *

Carnegie Mellon University
Language Technologies Institute
5000 Forbes Avenue, Pittsburgh, PA, USA 15213
(dhuggins,mohitkum,archan,awb,rkm,air)@cs.cmu.edu

ABSTRACT

The availability of real-time continuous speech recognition on mobile and embedded devices has opened up a wide range of research opportunities in human-computer interactive applications. Unfortunately, most of the work in this area to date has been confined to proprietary software, or has focused on limited domains with constrained grammars. In this paper, we present a preliminary case study on the porting and optimization of CMU SPHINX-II, a popular open source large vocabulary continuous speech recognition (LVCSR) system, to hand-held devices. The resulting system operates in an average 0.87 times real-time on a 206MHz device, 8.03 times faster than the baseline system. To our knowledge, this is the first hand-held LVCSR system available under an open-source license.

1. INTRODUCTION

Mobile, embedded, and hands-free speech applications fundamentally require continuous, real-time speech recognition. For example, an intelligent, interactive personal information assistant where natural speech has replaced the cumbersome stylus input and cramped graphical user interface of a PDA. Many current applications, such as speech control of GPS navigation systems and speech-controlled song selection for portable music players and car stereos also require a reliable and flexible speech interface. Finally, sophisticated natural language applications such as handheld speech-to-speech translation[1] require fast and lightweight speech recognition.

Several technical challenges have hindered the deployment of such applications on embedded devices. The most difficult of these is the computational requirements of continuous speech recognition for a medium to large vocabulary scenario. The need to minimize the size and power consumption for these devices leads to compromises in their hardware and operating system software that further restrict their capabilities below what one might assume from their raw CPU speed. For example, embedded CPUs typically lack hardware support for floating-point arithmetic. Moreover, memory, storage capacity and bandwidth on embedded devices are also very limited. For these reasons, much of past work (e.g. [2], [3]) has concentrated on simple tasks with restrictive grammars.

*This research was supported by DARPA grant NB CH-D-03-0010. The content of the information in this publication does not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

In addition to hardware limitations, interested developers face a high barrier in building such systems. It requires access to proprietary speech recognition toolkits which are often expensive and usually provided without source code. As well, popular embedded operating systems may lack many of the features developers take for granted on modern desktop systems, most notably a complete standard C/C++ programming library and a fast virtual memory subsystem.

POCKETSPHINX is the authors' attempt to address the above issues. Our work builds on previous research in the Carnegie Mellon Speech group related to fast search techniques ([4] and [5]) and fast GMM computation techniques ([6], [7] and [8]). We believe that this work will benefit the development community and lead to the easier creation of interesting speech applications. Therefore, we have made this work available to the public without cost under an open-source license. To the best of our knowledge, this is the first open-source embedded speech recognition system that is capable of real-time, medium-vocabulary continuous speech recognition.

2. BASELINE SPHINX-II SYSTEM

The target hardware platform for this work was the Sharp Zaurus SL-5500 hand-held computer. The Zaurus is typical of the previous generation of hand-held PCs, having a 206MHz StrongARM® processor, 64MB of SDRAM, 16MB of flash memory, and a quarter-VGA color LCD screen. We chose this particular device because it runs the GNU/Linux® operating system, simplifying the initial port of our system. However, the CPU speed and memory capacity of this device are several years behind the current state of the art, making it commensurately more difficult to achieve the desired level of performance. To build our system, we used a GCC 3.4.1 cross-compiler built with the `crosstool` script[1].

Platform speed directly affected our choice of a speech recognition system for our work. Though all the members of the SPHINX recognizer family have well-developed programming interfaces, and are actively used by researchers in fields such as spoken dialog systems and computer-assisted learning, we chose the SPHINX-II recognizer[2] as our baseline system because it is faster than other recognizers currently available in the SPHINX family.

To evaluate our system's performance, we used 400 utterances randomly selected from the evaluation portion of the DARPA Re-

[1] http://kegel.com/crosstool/
[2] http://www.cmusphinx.org/

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE **Pocketsphinx: A Free, Real-Time Continuous Speech Recognition System for Hand-Held Devices**	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Language Technologies Institute,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**4**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

source Management (RM-1) corpus. The acoustic model uses Hidden Markov Models with a 5-state Bakis topology and semi-continuous output probabilities. It was trained from 1600 utterances from the RM-1 speaker-independent training corpus, using 256 tied Gaussian densities, 1245 tied Gaussian Mixture Models (senones), and 39570 context-dependent triphones. The input features consisted of four independent streams of MFCC features, delta and delta-delta MFCCs, and power. A bigram statistical language model was used with a vocabulary of 994 words and a language weight of 9.5. The test set perplexity of this language model is 50.86.

On our development workstation, a 3GHz Intel® Pentium® 4 running GNU/Linux, SPHINX-II runs this task in 0.06 xRT. After the first stage of porting the system to the Zaurus target platform, without applying any optimizations, this same task takes 7.15 xRT. The baseline word error rate is 9.73%. Clearly, this is much too slow to be useful for even the simplest recognition tasks.

3. PLATFORM OPTIMIZATIONS

In the next stage of development, we investigated potential speed-ups based on our knowledge of the hardware platform. First, we noted that for embedded devices, memory access is slow and RAM is at a premium. We made several changes to the system to address this problem, described in Section 3.1. Second, the data representation was not optimal for the capabilities of the target CPU. Lastly, some implementation details led to inefficient code being generated for the target platform. The changes we made to address these issues are described in Section 3.2.

3.1. Memory Optimizations

Memory-mapped file I/O: For embedded devices, where RAM is a scarce resource, acoustic model data should be marked as read-only so that it can be read directly from ROM. On embedded operating systems, the ROM is usually structured as a filesystem, and thus it can be accessed directly by using memory-mapped file I/O functions such as `mmap(2)` on Unix or `MapViewOfFile()` on Windows.

Byte ordering: Unfortunately, the original binary formats for SPHINX-II acoustic and language models were not designed with read-only access in mind. In particular, they used a canonical byte-order that requires them to be read into memory and then byte-swapped. We modified the HMM trainer, SPHINXTRAIN, to output these files in the target system's native byte-order. We then modified SPHINX-II to use existing header fields to determine the byte-ordering of the file, thus allowing memory-mapping for files in the native byte order.

Data alignment: Modern CPUs either require or strongly encourage aligned data access. For example, a 32-bit data field is required to be aligned on a 4-byte address boundary. Where the model file formats mixed data fields of different widths, it was necessary to insert padding to ensure proper alignment. The result is that, while our version can read model files from previous versions, the files generated by it are not backward-compatible.

Efficient representation of Triphone-senone mapping: Generally in SPHINX-II these are read from two large text files, and stored in equally large hash tables. By contrast, the SPHINX-III system uses a single, compact "model definition" file which is represented by a tree structure in memory, a much more memory-efficient solution. Therefore, we back-ported the model definition code from SPHINX-III to our system, producing a significant reduction in memory consumption and a much faster startup time.

3.2. Machine-Level Optimizations

Use of Fixed Point Arithmetic: The StrongARM processor has no hardware support for floating-point operations. Floating-point computations must be emulated in software, usually by a set of math routines provided by the compiler or by the runtime library. Since these routines must exactly replicate the functionality of a floating-point coprocessor, they are too slow for arithmetic-intensive tasks such as acoustic feature extraction and Gaussian computation. Therefore, we found it necessary to rewrite all time-critical computation using integer data types exclusively.

Two basic techniques for doing this exist: either values can be kept pre-scaled by a given factor (usually a power of two, for the best performance) or they can be converted to logarithms (usually with a base very close to 1.0, for the best accuracy). The choice depends primarily on the dynamic range of the values in question and on the types of operations that will be performed on them. In our system, we calculate the Fast Fourier Transform (FFT) using signed 32-bit integers with a radix point at bit 16, that is, in Q15.16 format. However, to calculate the MFCC features, we need to take the power spectrum, whose dynamic range far exceeds the limits of this format. Since we will eventually take the log of the spectrum in order to compute the cepstrum, we use a logarithmic representation for the power spectrum and the Mel-filter bank. Addition of logarithms is accomplished using a lookup table, shared with the GMM computation component, which also operates on integer values in log-space.

The use of fixed-point arithmetic inevitably involves some rounding error, which is compounded by each operation performed. It is therefore important to choose algorithms that minimize the number of operations, not only for speed, but also to maintain accuracy. For example, one way to optimize an FFT for real-valued input data is to perform a half-length complex FFT on the input data, then post-process the output to separate the real and imaginary parts[9]. However, when this is done in fixed-point, the added processing leads to errors that can significantly increase the word error rate, in some cases by up to 20% relative.

Optimization of data and control structures: The ARM architecture is heavily optimized for integer and Boolean computation. Most instructions include a "shift count" field that allows the output operand to be bit-shifted by an immediate value without penalty. In addition, most instructions can be nullified on any condition, allowing many short branches to be eliminated. Finally, the ARM is a 32-bit load-store architecture with 16 general-purpose registers. Therefore it is important to keep data in registers while performing intensive computations, it is always faster to access memory 32 bits at a time, and unaligned accesses must be avoided at all costs. In general, a good optimizing compiler can make efficient use of the register file, but in some cases it is necessary to manually unroll loops in order to generate the most efficient code.

As a case in point, a large percentage of CPU time in the SPHINX-II system is spent in the maintenance of the list of active senones to be computed for each frame of input. In the baseline system, this list is generated by setting flags in an array of bytes which is then scanned to produce an array of senone IDs. This arrangement is sensible since there are typically many fewer active senones than total senones. However, the byte-array representation places greater load on the processor's cache, and also involves byte-wide accesses that are slow on CPUs such as ARM and PowerPC. Therefore, we changed the representation to be a bit vector, and unrolled the loop that scans this bit vector to operate on one 32-bit word at a time. This also allows it to skip entire blocks of 32 senones in the case where the active list is very sparse. In examining the generated assembly

code, we find that it is now very efficient: only 4 instructions are used to check each senone in the bit vector and conditionally add it to the list.

4. ALGORITHMIC OPTIMIZATIONS

After completing the platform and software optimizations detailed in the last section, we examined the output of the GNU `gprof` source-level profiling tool to determine where to look for more principled speed-ups. We found that the bulk of computation was spent in four areas: acoustic feature (MFCC) calculation, Gaussian (codebook) computation, Gaussian mixture model (senone) computation, and HMM evaluation (Viterbi search)[3]. The approximate proportion of time spent in these four areas is shown in Table 1.

Component	Desktop	Embedded
Codebook	27.43%	24.59%
HMM	24.68%	22.11%
MFCC	14.39%	11.51%
Senone	7.67%	11.71%

Table 1. Percentage of time spent in selected tasks

In our algorithmic optimizations, we concentrated primarily on Gaussian mixture model (GMM) computation, since in previous work [6] we have developed a well-reasoned framework for approximate GMM computation. In this framework, GMM evaluation is divided into 4 layers of computation:

1. *Frame layer:* all GMM computation for an input frame.
2. *GMM layer:* computation of a single GMM.
3. *Gaussian layer:* computation of a single Gaussian.
4. *Component layer:* computation related to one component in the feature vector (assuming a diagonal covariance matrix).

This framework allows a straightforward categorization of different speed-up techniques by the layer(s) on which they operate, and allows us to determine how different techniques can be applied in combination with each other. However, this framework, as with much other work in approximate GMM calculation, applies primarily to systems using continuous distribution HMMs (CDHMM). In application of the idea to semi-continuous HMMs (SCHMM), several differences should be noted:

- In the full computation of semi-continuous acoustic models, a single "codebook" of Gaussian densities is shared between all mixture models.
- The number of mixture Gaussians is usually 128 to 2048, which is much larger than the 16 to 32 densities used for each mixture in a typical CDHMM system.
- SCHMM-based systems usually represent the feature vector with multiple independent streams.

Therefore, though the four-layer model still applies, the layers beneath the frame layer are structured differently. In particular, since the codebook is shared between all GMMs, the entire codebook must be computed at every frame. This limits the degree to which approximations at the GMM layer can reduce computation. We applied the following techniques to each layer:

- **Frame layer:** We applied frame-based downsampling (see [10]). Although this inevitably results in a loss of accuracy, it is the only way we found to achieve a speed-up above the Gaussian layer.
- **GMM layer:** We attempted to apply context-independent GMM-based GMM selection ([11], [7]). However, we found that the overhead of this technique far outweighed the reduction in GMM computation.
- **Gaussian layer:** We considered several possibilities such as Sub-VQ-based Gaussian selection[8], but all of these involve significant overhead. Therefore, we decided to use a fast tree-based approach to Gaussian selection.
- **Component layer:** SPHINX-II already implements a form of partial Gaussian computation[12]. We used information from the tree-based Gaussian selection to improve the efficiency of this algorithm.

The computation of the codebook is already relatively quick in SPHINX-II. At each frame, the previous frame's top-N scoring codewords (for some small N, typically 2 or 4) are recomputed and the resulting distances are used as a threshold for partial computation of the remaining codewords[12]. Computation of senones has also already been optimized, by transposing the mixture weight arrays in memory and quantizing them to 8-bit integer values[4], as well as by providing separately optimized functions for each of the most common top-N values.

In the frame layer, we initially applied frame-based downsampling in a straightforward manner, by simply skipping all codebook and GMM computation at every other frame. However, we later modified this to recompute the top-N Gaussians from the previous frame and use these to compute the active senones from the current frame. This is analogous to the "tightening" of the CI-GMM selection beam that is implemented in SPHINX-III 0.6 [7]. We found that this was actually faster by a very small margin (0.6%) and also resulted in a 10% relative decrease in the word error rate.

In the Gaussian layer, we applied a modified version of the bucket box intersection algorithm, as described in [13]. This algorithm organizes the set of Gaussians in a kd-tree structure which allows a fast search for the subset of Gaussians closest in the feature space to a given feature vector. For each acoustic feature stream in the codebook, we build a separate tree of arbitrary depth (typically depth 8 or 10, to reduce storage requirements) with a given relative Gaussian box threshold. At each frame, after computing the previous frame's top-N codewords, we search the kd-tree to find a shortlist of Gaussians to be partially computed.

Though the trees are built off-line, the depth to search in the tree can be controlled as a parameter to the decoder at run-time. This allows the memory requirement for the trees to be quite small, since the shortlists of Gaussians need only be stored at the leaf-nodes. We also explored the idea of limiting the maximum number of Gaussians to be searched in each leafnode. In order to make this feasible, we sorted the list of Gaussians in a leafnode by their "closeness" to the k-dimensional region, or bucket box, delimited by that node. We found that an appropriate criterion is the log-ratio of the total volume of the individual Gaussian's bucket box to the area in which it overlaps with the leafnode's bucket box.

5. EXPERIMENTAL RESULTS

The result of applying various optimizations in sequence is shown in Table 2. As expected, the largest speed gain we were able to achieve

[3]In fact, 74% of the total running time is spent in only 13 functions!

	xRT	Speed-up	WER	Δ
Baseline	7.15	0	9.73%	0
w/Fixed-point	2.68	2.67	10.06%	+3.4%
w/Real FFT	2.55	1.05	10.06%	0
w/Log Approx	2.29	1.11	10.75%	+6.9%
w/Assembly	1.60	1.43	10.69%	-0.6%
w/Top-2 Gaussians	1.40	1.14	11.57%	+8.2%
w/Viterbi-only	1.06	1.32	12.61%	+8.9%
w/Downsampling	1.00	1.06	13.29%	+5.4%
w/Beam-tuning	0.89	1.12	14.61%	+9.9%
w/kd-trees	0.87	1.02	13.95%	-4.5%

Table 2. Performance and accuracy on 994-word RM task after successive optimizations

came from the use of fixed-point arithmetic. Simply reimplementing the existing acoustic feature extraction in fixed-point resulted in a 2.7-fold gain in speed. The loss of precision caused a slight increase in the word error rate, from 9.73% to 10.06%.

The only algorithmic "free lunch" came from the use of a FFT algorithm specialized for real-valued inputs[9]. After speeding up the FFT, the fixed-point logarithm function used in MFCC calculation became a bottleneck, so we reduced its precision, resulting in a significant gain in speed, albeit with a reduction in accuracy. We then reimplemented the fixed-point multiply operation using inline ARM assembly language, giving another large boost in speed with no degradation in accuracy.

We changed several decoder parameters to their "faster" values in order to boost the speed of the system. The number of top-N Gaussians used to calculate the senone scores was reduced to 2 (and a loop in the top-2 senone computation function was unrolled). SPHINX-II uses a multi-pass decoding strategy, performing a fast forward Viterbi search using a lexical tree, followed by a flat-lexicon search and a best-path search over the resulting word lattice. In order to get the best performance, we disabled the latter two passes.

Next, we applied partial frame downsampling, then used a separate held-out set of 200 utterances to find the optimal widths for the various beams used in decoding, in order to reduce the amount of time spent in Viterbi search. We then used the same held-out set to find the optimal threshold and depth of kd-trees to use. To our surprise the use of kd-trees actually reduced the error rate slightly.

The final system has a word error rate of 13.95% on our test set, degraded by 43.4% relative to the baseline system. We are encouraged by the fact that the largest sources of degradation were not related to our algorithmic optimizations, but rather from overly zealous tuning of the search parameters. Such tuning could be relaxed for more recent processors or can be adjusted for different tasks. It is also likely that with better acoustic modeling and cross-validation, these errors could be reduced or eliminated. In addition, this final system exhibited an 8-fold reduction in CPU usage from the baseline system and a 3-fold reduction from the baseline machine-optimized system.

6. CONCLUSION AND FUTURE WORK

In this paper, we present a 1000-word vocabulary system operating at under 1 xRT on a 206 MHz hand-held device. The system in question has been released as open source code and is available at `http://www.pocketsphinx.org/`. POCKETSPHINX inherits the easy-to-use API from SPHINX-II, and should be useful to many other developers and researchers in the speech community.

In future, we will apply this system to a task with a higher perplexity language model and larger vocabulary. A candidate for further optimization is the Viterbi search algorithm, which we have not discussed in depth in this paper. Such a system will support development of additional, more interesting applications. We are also working on a port of POCKETSPHINX to the popular Windows®CE operating system and Pocket PC hardware.

7. REFERENCES

[1] A. Waibel, A. Badran, A. W Black, R. Frederking, D. Gates, A. Lavie, L. Levin, K. Lenzo, L. Mayfield Tomokiyo, J. Reichert, T. Schultz, D. Wallace, M. Woszczyna, and J. Zhang, "Speechalator: Two-way speech-to-speech translation in your hand," in *Proceedings of NAACL-HLT*, 2003.

[2] H. Franco, J. Zheng, J. Butzberger, F. Cesari, M. Frandsen, J. Arnold, V. R. R. Gadde, A. Stolcke, and V. Abrash, "Dynaspeak: SRI's scalable speech recognizer for embedded and mobile systems," in *Proceedsings of HLT*, 2002.

[3] T. W. Köhler, C. Fügen, S. Stüker, and A. Waibel, "Rapid porting of ASR-systems to mobile devices," in *Proceedings of Interspeech*, 2005.

[4] M. K. Ravishankar, *Efficient Algorithms for Speech Recognition*, Ph.D. thesis, Carnegie Mellon University, May 1996.

[5] M. Ravishankar, "Some results on search complexity vs accuracy," in *DARPA Speech Recognition Workshop*, 1997.

[6] A. Chan, J. Sherwani, M. Ravishankar, and A. Rudnicky, "Four-layer categorization scheme of fast GMM computation techniques in large vocabulary continuous speech recognition systems," in *Proceedings of ICSLP*, 2004.

[7] A. Chan, M. Ravishankar, and A. Rudnicky, "On improvements of CI-based GMM selection," in *Proceedings of Interspeech*, 2005.

[8] M. Ravishankar, R. Bisiani, and E. Thayer, "Sub-vector clustering to improve memory and speed performance of acoustic likelihood computation," in *Proceedings of Eurospeech*, 1997, pp. 151–154.

[9] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus, "Real-valued fast fourier transform algorithms," *IEEE Transactions on Acoustic, Speech, and Signal Processing*, vol. 35, no. 6, pp. 849–863, 1987.

[10] M. Woszczyna, *Fast Speaker Independent Continuous Speech Recognition*, Ph.D. thesis, Universität Karlsruhe; Institut für Logik, Komplexität und Deduktionssysteme, 1998.

[11] A. Lee, T. Kawahara, and K. Shikano, "Gaussian mixture selection using context-independent HMM," in *Proceedings of ICASSP*, 2001, vol. 1, pp. 69–72.

[12] B. Pellom, R. Sarikaya, and J. H. L. Hansen, "Fast likelihood computation techniques in nearest-neighbor based search for continuous speech recognition," *IEEE Signal Processing Letters*, vol. 8, no. 8, pp. 221–224, July 2001.

[13] J. Fritsch and I. Rogina, "The bucket box intersection (BBI) algorithm for fast approximative evaluation of diagonal mixture Gaussians.," in *Proceedings of ICASSP*, 1996, pp. 837–840.